APMA 1940X: Information and Coding Theory
Project 2

Albert Caputo

March 14 2018

Markov Chain Monte Carlo Decoding

We are provided with a substitution cipher text such that each symbol, in this case letters of the alphabet, maps to a different unique letter in the alphabet. Let f denote this mapping such that

$$f : \text{True Letter} \rightarrow \text{Encrypted Letter}$$

encrypts the contents of the message. f is both one-to-one and onto the whole English alphabet including space. Our goal is to figure out the true contents of the message by finding f^{-1}. Searching the entire space of f's is computationally infeasible, so we wish to use probabilistic methods to speed up the process.

Using Markov Chain Monte Carlo (MCMC), we can decode the message probabilistically. First, we need to achieve a good estimate of the probability of English letters. That is, we seek to find the probability of a letter occurring given the letter preceding it. Denote this matrix M such that

$$M(x_i, x_{i+1}) = \mathbb{P}(X_{i+1} = x_{i+1} | X_i = x_i)$$

$$\sum_j M_{ij} = 1$$

where M is the probability transition matrix of English 2-grams. We can create M by mining a large English text. In our case, we use *War and Peace* by Leo Tolstoy.

Now we upload our cipher text, which looks like this:

ioiworipmin pifoyycwiovwindcmci ...

This of course looks completely meaningless at first, but using our transition Matrix M, we can make good guesses at f^{-1} and keep choosing better and better guesses until we have the correct one.

Implementing the MCMC Algorithm

Once we have M, we can construct the likelihood function L for any given guess g of f^{-1}. Let

$$L(g) = \sum_{i=1}^{n-1} \log\bigl(1 + M(g(x_i), g(x_{i+1})\bigr)$$

where n is the length of the entire cipher text. $L(g)$ sums over each pair in the cipher text under the substitution g. Note that we use the logarithm to avoid numerical underflow problems. Furthermore, we add 1 to each transition probability to avoid $-\infty$ in our calculations. These modifications still yield a function that strictly increases as our guesses g get closer to the true value of f^{-1}.

Now we have the tools we need to construct our MCMC algorithm. We first make an initial good guess at f^{-1}. How can we make a good guess? Well, we can assume that space has the highest probability (roughly 18% in *War and Peace*) of occurring out of any symbol. Thus, let's calculate the probability of each character occurring in the cipher text. We find that "i" has a probability of roughly 18%, similar to our sample text. So to initialize g^0, let's swap "i" and space, leaving every other letter mapping to itself.

Continuing, our algorithm will look like this:

1. Start with preliminary guess g.
2. Compute $L(g)$.
3. Pick two letters at random from g and swap them to form g^*.
4. Compute $L(g^*)$.
5. If $L(g^*) > L(g)$:
   (a) Let $g = g^*$.
6. Else if $L(g^*) < L(g)$:
   (a) Compute a random $u \sim$ Uniform(0,1).
   (b) If $u < \bigl(\frac{L(g^*)}{L(g)}\bigr)^\alpha$ for some fixed scaling parameter α, let $g = g^*$.
   (c) Else, Stay at g.
7. Repeat Steps 2-6 until convergence (somewhere between 10,000 and 30,000 iterations).

Remarkably, this algorithm works very well in finding f^{-1} quickly. We are usually choosing better and better guesses g, but use step 6 as a safety measure by choosing a worse guess with small probability to avoid local maximums. The scaling parameter α is very important in that a low value will cause many jump downs, which is sub-optimal since we may never find f^{-1}. On the other hand, if we choose α to be too large, we won't jump down often enough and get stuck at a sub-optimal guess. Using the choice of our likelihood function, it is often the case that $L(g)$ and $L(g^*)$ are very close together. After many test runs, α is assumed to be best fixed somewhere between 200-300 ($0.99^{300} \approx 0.05$). Note that we keep track of each guess g and only keep the best guess after the algorithm is finished i.e.

$$g^{**} = \underset{g}{\operatorname{argmax}}\, L'(g)$$

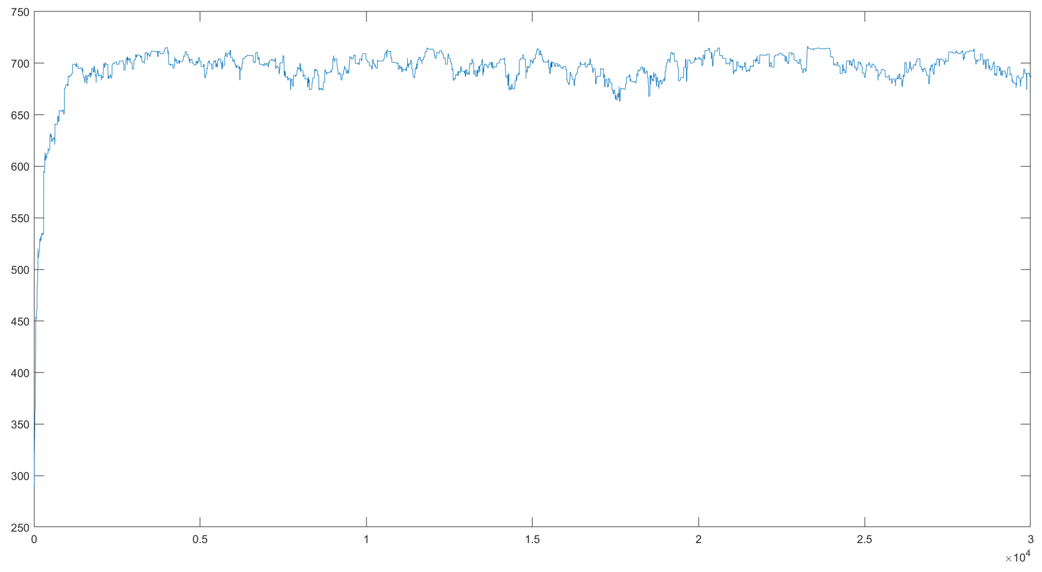
where $L'(g) = \{L(g^0), L(g^1), ...\}$. Ideally, we want $g^{**} = f^{-1}$ at the end of every MCMC run. Usually this is the case, but sometimes, we only get very close, such as having all letters correct except mapping to "x" instead of "j" and vice versa since there are only one of each of these letters in the decoded text. Some other common mistakes are swapping "m" and "p" as well as "b" and "j."

Here is an example of our algorithm running for 30,000 iterations, printing the output of $g(C)$ where C is our code and g is our best guess so far. We print the max $g(C)$ so far every 500 iterations, but for the sake of conciseness any $g(C)$ without changes from the previous best guess are omitted:

```
a gaf un twu raddeg asg thene wad oneat altizitf aquang the recxug sut usmf
a daf us twu bayyed and these way gseat activitf aquasd the bezoud nut unlf
a daf or two mayyed and there way great activitf ajoard the meqbod not onlf
a das or two payyed and there way great activits aboard the pequod not onls
a day or two kassed and there was great activity aboard the kequod not only
a day or two passed and there was great activity ajoard the pequod not only
a day or two passed and there was great activity aboard the pequod not only
```

As we can see, the first guess makes no sense, but slowly throughout each iteration we get closer to the true message, which we find is an excerpt from *Moby Dick* by Herman Melville. This particular run happened to successfully find the correct output, avoiding any aforementioned small mistakes.

To achieve a more detailed understanding of how the algorithm is behaving from iteration to iteration, we can plot the likelihood of every guess so far from the same run:



here, the x-axis represents the current iteration and the y-axis represents the likelihood. With the efficient initialization of g, it is interesting to see how $L(g)$ quickly shoots up to around 700 (about 2000 iterations), but takes a long time to find f^{-1} after reaching this plateau. This is because the algorithm keeps trying to break through local maxima using the step-6 protocol, giving way to the jagged, yet steady appearance. The maximum was achieved at roughly the 23000th iteration.

Possible Improvements

We have shown how effective MCMC is at decoding substitution ciphers. Although the algorithm works, we must ask the question, can we do better? 30,000 iterations is a solid amount of computational work and even then the algoirthm still occasionally fails to converge to the true solution.

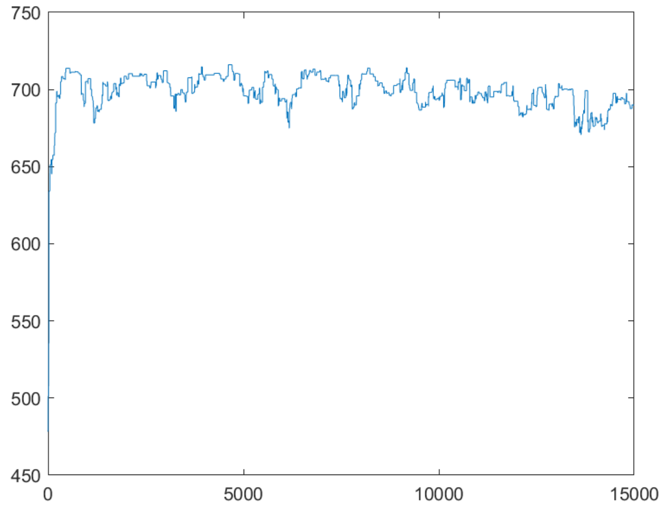
An easy trick to improve run time is to improve our initial guess g^0. So far we only assume which character maps to space using the probability distribution of our cipher. Applying this same logic, we can order each character in the cipher in descending order according to frequency, do the same with our sample text, and align each. Then, we initialize g such that each code letter c will map to the closest letter in sample probability l. Hence,

$$c = \{c_1, ..., c_{27}\}$$
$$p(c_1) > p(c_2) > ... > p(c_{27})$$

$$l = \{l_1, ..., l_{27}\}$$
$$p(l_1) > p(l_2) > ... > p(l_{27})$$
$$g^0(c_i) = l_i \qquad \forall i \in \{1, ..., 27\}$$

Under this initialization, we can cut our run time roughly in half by iterating 15,000 times and achieve the same results as before.

Plotting likelihood after a 15,000 iteration run of this new initialization yields



where g converges to the true f^{-1} at roughly the 5000th iteration (this is lucky, usually converges at around the 10,000th iteration). Note that we still occasionally do not converge to the true f^{-1}, but we almost always get very close or better.

MATLAB Code and Explanations

To implement MCMC, we first used MATLAB to parse through our sample text and keep track of each 2-gram transition probability. Since 27×27 is fairly small, we implemented this as an actual matrix. There are some preprocessing steps involved before calculating M. First, we convert the text to ascii values using the same code from Project 1:

```
function [output] = convert2Ascii(text)
%CONVERT2ASCII converts a .txt file to ascii, removing punctuation and
%capitalization, labelling a,...,z,' ' as 1-27.
label = fopen(text);
[hk,count] = fread(label,'uchar');
```

```
fclose(label);
F = double(hk);
F(F<33) = 27;
F(F>64 & F<91) = F(F>64 & F<91) - 64;
F(F>96 & F<123) = F(F>96 & F<123) - 96;
F = F(F<28);
F2 = [0;F(1:(size(F)-1))]+F;
output = F(F2<54);
end
```

Next, we create a rolling window matrix from our ascii-converted text such that the ith row contains the i^{th} and $i+1^{th}$ letters, which visually looks like this:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 5 \\ 5 & 6 \\ \vdots & \vdots \end{bmatrix}$$

Which we can do easily in MATLAB through using the hankel() function:

```
function output = createRollingWindow(vector, n)
% CREATEROLLINGWINDOW returns successive overlapping windows onto a vector
%   OUTPUT = CREATEROLLINGWINDOW(VECTOR, N) takes a numerical vector VECTOR
%   and a positive integer scalar N. The result OUTPUT is an MxN matrix,
%   where M = length(VECTOR)-N+1. The I'th row of OUTPUT contains
%   VECTOR(I:I+N-1).
l = length(vector);
m = l - n + 1;
output = vector(hankel(1:m, m:l));
end
```

Now we can compute M. To do this, we first find each row in the rolling window matrix starting with i corresponding to the i^{th} symbol in our alphabet. We then compute the probability of each next symbol j occurring given that the first symbol i occurred, i.e. tabulating the frequency of each member of the second column given that the symbol in the first column is i. We then divide by the sum of all occurrences and store each value in the j^{th} column of the i^{th} row of the transition matrix. We do this for all $i \in \{1, ..., 27\}$. Thus, in MATLAB,

```
function [tranMatrix] = createTran(text)
%createTran generates the transition matrix of the sample text.

tranMatrix = zeros(27);

for i = 1:27
```

```
        textI = text(find(text(:,1) == i),:);
        M = tabulate(textI(:,2));
        tranMatrix(i,M(:,1)) = M(:,2)./sum(M(:,2));
end
```

Note that here M is a placeholder while tranMatrix represents the aforementioned M.

Lastly, we construct a function to compute a step of MCMC. Note that we convert the cipher text to a rolling window as well. This algorithm is identical to the one previously explained. In MATLAB,

```
function [LL,LLs,fnew] = mcmcStep(f,alpha,tranMatrix,text)
%MCMC takes as input the current guess f, scaling parameter alpha, the
%transition matrix, and the text.
%   Outputs either a new guess or stays at the current guess based on MCMC
%   protocol. Also outputs each likelihood function.

% Choose random indeces to flip
fs = f;
idx = randi(length(f),1,2);
fs(idx) = fs(flip(idx));

% Initialize current LL and new LLs
LL = 0;
LLs = 0;

% Compute LL and LLs
for i = 1:size(text,1)
        LL = LL + log(1+tranMatrix(f(text(i,1)),f(text(i,2))));
        LLs = LLs + log(1+tranMatrix(fs(text(i,1)),fs(text(i,2))));
end

% Choose whether or not to transition
if LLs > LL
        fnew = fs;
elseif rand(1)<(LLs/LL)^alpha
        fnew = fs;
else
        fnew = f;
end
end
```

We have all the tools we need to put it all together and choose our initial g^0, scaling parameter α, and the number of iterations. Finally, we print the progress of the algorithm and plot the likelihood at each step:

```
% Code from Pattern Theory:
```

```
% Converts .txt to ascii
% Upload Training Data: War and Peace
G = convert2Ascii('WAP.txt');
G2 = createRollingWindow(G,2);

% Compute Transition Matrix M
M = createTran(G2);

% Upload Cipher
C = convert2Ascii('scram2G.txt');
C2 = createRollingWindow(C,2);

% Initialize Params
iter = 15000;
LL = zeros(iter,1);
LLs = zeros(iter,1);
f = zeros(iter+1,27);

% Start with 1-to-1
f(1,:) = 1:27;

% Good guess swapping for correct space
%f(1,[9 27]) = f(1,[27 9]);

% Better guess using entire prob dist
pC = tabulate(C);
pG = tabulate(G);

[a_sorted, a_order] = sort(pC);
pC = pC(a_order);

[b_sorted, b_order] = sort(pG);
pG = pG(b_order);

pC = fliplr(pC(:,3)');
pG = fliplr(pG(:,3)');

for i = 1:27
        f(1,probIdx(1,i)) = probIdx(2,i);
end
probIdx = [pC;pG];

% Iterate
for k = 1:iter
        [LL(k),LLs(k),f(k+1,:)] = mcmcStep(f(k,:),300,M,C2);
        % Print max so far every 500 steps
        if mod(k,500) == 0
                [~,ind] = max(LL);
                fm = f(ind+1,:);
                D = fm(C(1:100));
                decode = strrep(char(D + 96),'{','␣')
    end
end

% Plot likelihood
plot(LL)
[~,ind] = max(LL);
fm = f(ind+1,:);
D = fm(C);
decode = strrep(char(D + 96),'{','␣')
```